1

Introducing Symfony, CakePHP, and Zend Framework

An invasion of armies can be resisted, but not an idea whose time has come.

— Victor Hugo

COPYRIGHTED MATERIAL

WHAT'S IN THIS CHAPTER?

- General discussion on frameworks.
- Introducing popular PHP frameworks.
- Design patterns.

Everyone knows that all web applications have some things in common. They have users who can register, log in, and interact. Interaction is carried out mostly through validated and secured forms, and results are stored in various databases. The databases are then searched, data is processed, and data is presented back to the user, often according to his locale. If only you could extract these patterns as some kind of abstractions and transport them into further applications, the development process would be much faster.

This task obviously can be done. Moreover, it can be done in many different ways and in almost any programming language. That's why there are so many brilliant solutions that make web development faster and easier. In this book, we present three of them: *Symfony*, *CakePHP*, and *Zend Framework*. They do not only push the development process to the extremes in terms of rapidity but also provide massive amounts of advanced features that have become a must in the world of Web 2.0 applications.

WHAT ARE WEB APPLICATION FRAMEWORKS AND HOW ARE THEY USED?

A web application framework is a bunch of source code organized into a certain architecture that can be used for rapid development of web applications. You can think of frameworks as half-produced applications that you can extend and form to make them take shape according to your needs. Well, that means half your work has already been done, but for some it is as much a blessing as a curse because this work was done in a particular way, without your supervision.

Thus all frameworks are either stained with a coding methodology and naming and structural conventions, or if they try to avoid these restrictions, they need to be heavily configured by you. This either reduces their flexibility or makes their learning curve significantly steeper. And if you really want to escape from these problems toward a more library-like approach, you have to sacrifice some development speed. You can see that frameworks are all about tradeoffs.

That's why it is really good to take a look at many frameworks and compare their differences. Perhaps one of them offers conventions that you would use as good practices, anyway? Perhaps you have nothing against some initial configuration that allows you to be rapid and flexible at the same time? And maybe you want just a library of powerful components to link together by yourself? The choice is yours, and if you find a way to mitigate their disadvantages, you can fully enjoy the greatest benefit of all frameworks: truly rapid development.

Further advantages of frameworks are elegance of code and minimizing the risk of programming errors. Frameworks conform to the Don't Repeat Yourself (DRY) principle, which means that they have all the pieces of logic coded only once in one place. This rule forbids duplication of code, especially *copypasting*. This facilitates maintenance of code and prevents nasty errors. Generally, frameworks promote code reusability and other good programming practices wherever they can, which is great for programmers who do not have enough knowledge or discipline to care for quality of code by themselves.

Another great feature is the clean organized look of links that can be done with URL rewriting, which is supported by most frameworks. Instead of `/animals.php?species=cats&breed=mainecoon`, type just `/animals/cats/mainecoon`. This is not only appealing to the eye but also very search engine optimization (SEO)–friendly.

Framework versus Library

The main difference between a library and a framework is that:

- *libraries* are called from your code
- *frameworks* call your code

In other words, a framework in your application is a skeleton that you fill with features or serves as a platform on which you build your modules. Whereas a library instead provides attachable modules on top of a platform made by yourself. Some people perceive a framework as something better or more complete than a library, so "framework" became a buzzword that is often overused. That's why people call some libraries *frameworks*, even though they do not invoke developers' code. There

is nothing wrong with a piece of code being a library, as it is just a different entity. And there are also some bad frameworks that damage the reputation of the good ones — basically you can take any half-done application, release it, and call it a framework. These two software groups just behave differently and should not be confused.

The application architecture utilized by frameworks is called *inversion of control*, because the data flow is inverted compared to ordinary procedural programming. It is also referred to as *The Hollywood Principle*: "Don't call us, we'll call you." This corresponds to third-party code calling developer's code. The main reason behind it is to make the high-level components less dependent on their subsystems. High-level components pass the control to low-level components, who themselves decide how they should work and when to respond. A good example is the difference between a command-line program, which stops and then asks the user for input, and a program with a windowed user interface, in which the user can click any button and then the window manager calls the program instead.

Some frameworks, such as Zend Framework or CodeIgniter, follow *loosely coupled* architecture, which means that their components are less dependent on each other and may be used separately, more library-style. Loosely coupled frameworks do not provide development as rapidly as those following a tighter framework architecture and Model-View-Controller (MVC) pattern; however, such an approach allows more flexibility and control over code.

When You Should Use a Framework and When You Should Not

Frameworks are not the cure for all programming problems. Putting aside today's awesome state of development, you should always remember how frameworks were created a few years ago. Most of them were more or less unoptimized junk created by one guy to help him speed up *his* development process, without much care for documentation, elegance, ease of use, or even readability of his code. Then another group of guys took this code and bloated it with a patchwork of extra functionalities barely consistent with the original code. Then it became apparent that this whole lot needs a solid cleanup in order to be usable, but this would mean either rewriting it from scratch or packaging code in additional wrapper classes, further increasing its unnecessary complexity.

Of course, today the disorganized origin of frameworks is not as evident as before because the quality of code has risen considerably. But still, that's why most beefed-up frameworks have performance issues. That's why they are not always easy to learn. And that's why new ones emerge to cover up weaknesses of older ones. And finally that's why major frameworks provide completely rewritten 2.0 versions, which address all previously mentioned problems.

Advantages

When web application frameworks are useful:

- For more or less standard projects with dynamic content, like social networking, online stores, news portals, and so on
- For easily scalable applications that can grow from start-up to worldwide popular services without need for big changes in code

- For producing consecutive apps, in which modularity and reusability of pieces of code like controllers and views may be helpful
- For real-world development with deadlines, rotating staff, and fitful customers
- If you are, or want to be, a professional web developer, so learning how to work with frameworks is not an excessive effort

As you can see, this applies to most commercial web applications that connect to a database and allow its users to create and modify its content. Therefore, programming with web app frameworks becomes a standard and common practice in the web development world.

Disadvantages

When you should consider development without any frameworks at all:

- Purely informative web pages without user-created content, for example an artist's portfolio with fancy graphics
- Small projects with limited database connection that wouldn't benefit much from frameworks' code generation
- Really big projects that additionally need extreme performance, like the Google suite (you would be using a compiled programming language for that rather than PHP, anyway)
- With limited hardware resources that call for top performance as well (not really a likely scenario because programming costs are now always higher than hardware costs)
- Specialist or experimental applications that may evolve in completely unknown direction or work with some custom solutions, like interfaces for scientific experiments with an object-oriented database
- When you really need (and can afford) total control over the code and evolution of the application
- When you want to create a web app, but you or your co-workers don't want or, even worse, cannot learn how to use a framework

These conditions are generally fulfilled by three types of projects: *small* static websites, extremely *specialist* websites, and *failed* websites. Frameworks are created for development of common web applications with well-known standard architecture. Of course, they may be greatly extended thanks to plug-ins and modules, but complete alteration of their structure may require much painful hacking, so you should always check their capabilities with the design requirements of your project.

PHP versus Other Programming Languages

PHP for many years has been a very popular programming language; however, it was commonly judged as unprofessional. A stereotypical PHP developer was an undereducated freelancer producing cheap, low-quality code. Professionals were supposed to use Zope, ASP, or various Java

technologies. Then in 2005 there was a boom of Ruby. Everyone was amazed with the elegance of this programming language; and *Ruby on Rails*, the central piece of software ever written in Ruby, was claimed to be the ultimate web applications framework. Soon clones of Ruby on Rails began popping out. That's how Python's Django and Turbogears, as well as all PHP frameworks were born.

In 2004 PHP5 was released. It was neat and object-oriented. If somebody still wrote old-styled HTML mixed with pieces of PHP script, it was only his choice, and the programming language no longer was to blame. It took some time, but people gradually considered PHP as a disciplined and professional tool. Together with the modern MVC paradigm and features styled after other frameworks, PHP begun its amazing way to the top of web development applications.

After a few years, it became evident that Ruby on Rails had various limitations. One important limitation was the low availability and high price of Ruby hostings while there was a lot of cheap hosting for PHP everywhere in the world. There was also a great community that eagerly developed early PHP frameworks. This resulted in an IT revolution that dethroned Ruby on Rails as the most popular framework and placed a council of PHP frameworks in its place.

Figure 1-1 illustrates the change in interest in various frameworks over time expressed as search volume in the Google search engine in the *Computers* & *Electronics* category. The figure was created with *Google Insights for Search*, which is a more advanced form of the well known *Google Trends* tool. You can check these search terms yourself to obtain results beyond mid-2010 (that's when this book was written), at the website `www.google.com/insights/search/`.

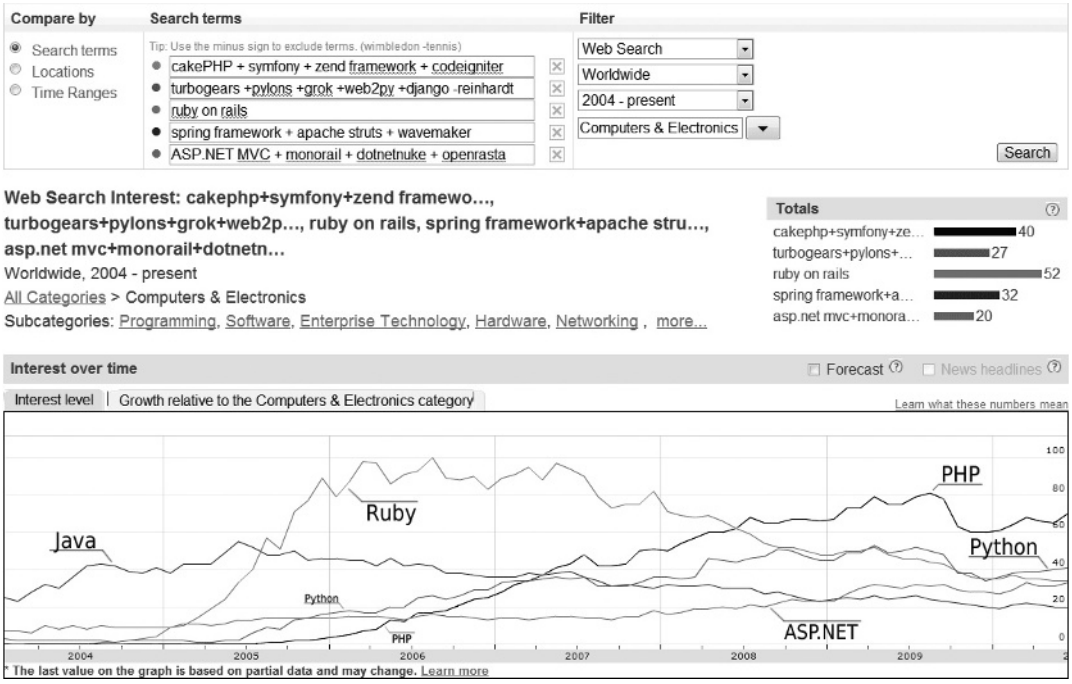

FIGURE 1-1: Search volumes of frameworks in various programming languages

OPEN SOURCE PHP WEB FRAMEWORKS

Another question we want to answer is why we have chosen these three particular frameworks. Are they really *better* in any way, or are we biased or perhaps have some financial interest in promoting them? Well, starting with that last question, we are completely independent open source enthusiasts and we wanted to compare free ("free" as free speech) software only, so there is certainly no *Evil Corporation* behind us, and nobody told us which frameworks to choose. We answer the question of whether they're better than other frameworks in the following sections.

There were once closed source PHP frameworks as well, but due to widespread success of the free frameworks, nowadays closed source frameworks are a thing of the past.

Comparison of Popular Interest

We have chosen Symfony, CakePHP, and Zend Framework due to their popularity in the web developers' community, including our own experience in PHP. We believe that open source programming tools show at least some correlation between their *popularity* and *quality* because they are used only if they are really useful. In that way they are different from things like proprietary software or pop music, in which quality can be easily replaced by aggressive *marketing* as the *popularity* gaining factor.

It turns out that the public interest in web frameworks can be measured quite objectively. Figure 1-2 shows search volumes for various PHP frameworks in *Google Insights for Search*. You can easily see that there are four leading competitors. All the others *combined* are less popular than *any one* of these four. The Lithium and Prado frameworks have been deliberately omitted because their names are nonunique, which generates *false positives* in trends. We have checked these names in specific categories and found that they are not significant as search terms, either.

When users search for information on a framework, the search results usually reflect talk about it on various blogs and forums, items about learning this technology, and finally developing applications using it. So public interest in a web framework results in real, long-term use of it.

CodeIgniter was really problematic for us. We had a long debate whether it should be included as one of the main frameworks. Perhaps now it is as frequently searched for as Symfony or CakePHP, but what matters more is the *area under the graph* because it reflects how many people have found the answers they sought and have probably used this knowledge for their projects.

Of course this graph shows nothing more than search volume, and when you see such fast growth it is hard to distinguish a long-lasting trend from temporary hype. We know that CodeIgniter is really good, so it is definitely more than a fad, and perhaps in a year or two it will have its place among the leading web tools.

We finally agreed that *three men* against *four frameworks* is not an equal fight. We have not completely forsaken CodeIgniter, though; its features are described, along with Lithium and Agavi, in Appendix B, where a simple application is developed using each one of them.

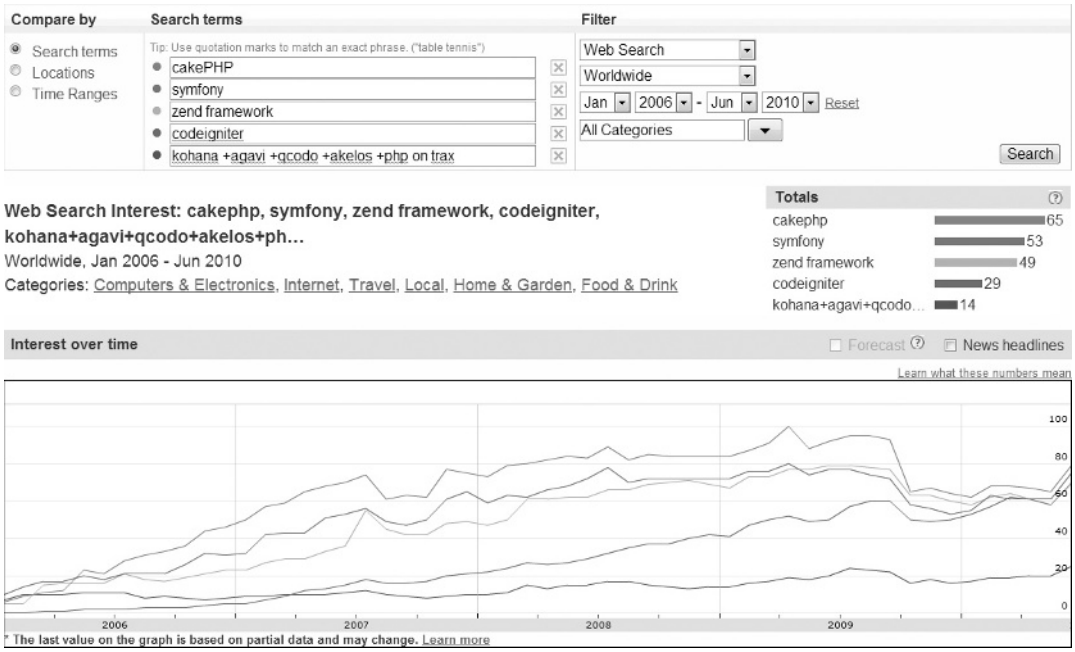

FIGURE 1-2: Comparison of search volumes of different PHP frameworks

The First Look

The first look at the frameworks really gives us little information on their individual features. Their websites just try to impress you with marketing descriptions and a list of features that vary little from one framework to another:

> *"**Symfony** is a full-stack framework, a library of cohesive classes written in PHP. It provides an architecture, components and tools for developers to build complex web applications faster. Choosing symfony allows you to release your applications earlier, host and scale them without problem, and maintain them over time with no surprise. Symfony is based on experience. It does not reinvent the wheel: it uses most of the best practices of web development and integrates some great third-party libraries."*

> *"**CakePHP** is a rapid development framework for PHP that provides an extensible architecture for developing, maintaining, and deploying applications. Using commonly known design patterns like MVC and ORM within the convention over configuration paradigm, CakePHP reduces development costs and helps developers write less code."*

> *"Extending the art & spirit of PHP, **Zend Framework** is based on simplicity, object-oriented best practices, corporate friendly licensing, and a rigorously*

tested agile codebase. Zend Framework is focused on building more secure, reliable, and modern Web 2.0 applications & web services."

Now see whether you can spot three differences. Well, the websites are not really informative about unique features of their frameworks. You can find more in various blogs and forums, but still there is little verified data, and general discussions tend to exchange purely personal opinions.

That is why we have written this book. In fact, the differences between frameworks are not really obvious, and it takes some time and practical examples to see them and then harness them in business solutions. Let's begin with some most basic facts.

Symfony

Started: 2005

License: MIT

PHP versions:

- Symfony 1.4: PHP 5.2.4+
- Symfony 2.0: PHP 5.3+

FIGURE 1-3: Symfony logo

Its logo is shown in Figure 1-3. Website: `www.symfony-project.org`

Symfony was produced in a French web development company, *Sensio Labs*, by Fabien Potencier. First it was used for the development of its own applications and then in 2005 it was released as an open source project. Its name was "symfony," but it is sometimes capitalized (as we do in this book) in order to make it more distinct.

Symfony was based on an ancient Mojavi MVC framework, with some inevitable influences from Ruby on Rails. It also integrated Propel Object-Relational Mapper and took advantage of the YAML Ain't Markup Language (YAML) serialization standard for configuration and data modeling. The default object-relational mapping (ORM) solution has been later changed to Doctrine.

Today Symfony is one of the leading web frameworks. It has a large active community and a lot of documentation — mainly free e-books. Symfony 2.0 is being released in late 2010. It offers various new features and greatly enhanced performance.

CakePHP

Started: 2005

License: MIT

PHP versions: 4.3.2+

Its logo is shown in Figure 1-4. Website: `http://cakephp.org`

FIGURE 1-4: CakePHP logo

CakePHP was started in 2005 by the effort of Polish web developer Michał Tatarynowicz. Heavily inspired by Ruby on Rails, CakePHP is an entirely community-driven open source project with lead developer Larry Masters (*aka* PhpNut). The next major release of CakePHP has also been announced, but its release date is still unknown.

The most important goals of CakePHP are its friendliness, development speed, and ease of use. And it really excels in that. Works out of the box (or oven), with no configuration. It has perfect documentation with working examples for most of its features. And it has really a lot of features to use. That allows the most rapid development with a smaller amount of code.

One of the most controversial features of CakePHP is its compatibility with PHP4. While once it allowed deployment on old cheap hosts that did not support PHP5, now it is more a drawback hindering CakePHP's development. Fortunately, version 2.0 will use PHP 5.3+. There are also reports of CakePHP's really bad performance, but they were mainly due to disabled caching by default.

Zend Framework

Started: 2005

License: new BSD

PHP versions: 5.2.4 since ZF 1.7.0

Its logo is shown in Figure 1-5. Website: `http://framework.zend.com`

FIGURE 1-5: Zend Framework logo

Zend Framework is sponsored by the U.S.-Israeli company, Zend Technologies Ltd., which was cofounded by Andi Gutmans and Zeev Suraski, the core developers of PHP. Strategic partners of Zend Technologies Ltd. include Adobe, IBM, Google, and Microsoft. The company offers various commercial products; however, Zend Framework is an open source project released under the "corporate friendly" *new BSD* license.

ZF is meant to be simple, component-based, and loosely coupled. This means that it is a *library of components*, which you can use as you wish, and usage of MVC architecture is optional. This lowers the learning curve and increases its flexibility. The documentation is great, and the source code is of very high quality, both because it's fully object oriented and thoroughly unit-tested. Zend announced an upcoming 2.0 version as well, but its release date is still unknown.

Other Frameworks

There are *hundreds* of PHP frameworks. This is not an exaggeration if you count all of them, including ancient and already abandoned projects, as well as brilliant younger startups and some useless short-lived junk. The web app market is a big one, but the amount of PHP tools is

disproportionally huge and perhaps somewhat excessive. Here is an overview of a few more notable ones that we have found to be used successfully to develop web applications.

CodeIgniter

Started: 2006

License: modified BSD

PHP versions: 4.3.2+

Its logo is shown in Figure 1-6. Website: `http://codeigniter.com`

FIGURE 1-6: CodeIgniter logo

CodeIgniter is developed and maintained by a privately-owned software development company, Ellis Labs. It is focused on having a very small footprint, while allowing a big increase in performance. It follows the MVC pattern only partially, for the models are optional. It is loosely coupled and in the words of Rasmus Lerdorf, it's "the least like a framework." Its lightweight approach has earned a wide recognition in the developers' community, but it is sometimes criticized for conformance with PHP 4.

CodeIgniter is a good choice for less complex web applications that would benefit from using a framework, but the heavier ones would either hinder the applications' performance with excessive features, or their configuration would take too much time. The structural simplicity of CodeIgniter makes it also a frequent pick by beginners who choose it as learning platform before moving to a full MVC framework.

Lithium

Started: 2009

License: BSD

PHP versions: 5.3+

Its logo is shown in Figure 1-7. Website: `http://lithify.me`

FIGURE 1-7: Lithium logo

Lithium took all the best that CakePHP had to offer and moved it to PHP 5.3. First it was a branch of CakePHP called Cake3, now it is a separate project run by some former CakePHP developers. It is lightweight, fast, and extremely flexible with extensive plug-in support. It has many truly experimental and innovative functions like a filter system and an integrated test suite.

The second search result Google showed us for "Lithium framework" is a page titled "CakePHP is dead…Lithium was born." This claim is still far from true, however, with the advantages provided by Lithium's support for PHP 5.3, Lithium may really endanger CakePHP in the future unless the latter takes immediate action.

Agavi

Started: 2005

License: LGPL

PHP versions: 5.2.0+ (recommended 5.2.8+)

Its logo is shown in Figure 1-8. Website: `www.agavi.org`

FIGURE 1-8: Agavi logo

Like Symfony, Agavi is based on the Mojavi framework. It was started in 2005, but the 1.0.0 version was worked upon until early 2009. The source code is very polished and sometimes called the best-written MVC OOP framework. However, it has not gained much popularity, perhaps due to scarce documentation.

It was never meant to be popular. The authors stress that Agavi is not a website construction kit, but a serious framework built with power and extensibility in mind. Its target applications are long-term specialist projects that need full control of their developers.

Kohana

Started: 2007

License: BSD

PHP versions: 5.2.3+

FIGURE 1-9: Kohana logo

Its logo is shown in Figure 1-9. Website: `http://kohanaphp.com`

Kohana is a community-supported offshoot of CodeIgniter. In contrast with CodeIgniter, Kohana is designed for PHP5 and is fully object oriented. While boasting higher elegance of code, it still has all the qualities of CodeIgniter: It is extremely lightweight, flexible, and easy to learn. The community behind Kohana is large and active, so despite its young age it should be considered a stable and reliable framework.

Prado

Started: 2004

License: revised BSD

PHP versions: 5.1.0+

FIGURE 1-10: Prado logo

Its logo is shown in Figure 1-10. Website: `www.pradosoft.com`

Prado stands for *PHP Rapid Application Development Object-oriented*. It enjoyed moderate popularity some time ago, but now its development seems a bit sluggish. However, it is still a mature framework well-suited for most business applications. One of its interesting features is that it nicely supports event-driven programming. It has some similarities with ASP.NET.

Yii

Started: 2008

License: BSD

PHP versions: 5.1.0+

Its logo is shown in Figure 1-11.
Website: `www.yiiframework.com`

FIGURE 1-11: Yii logo

Yii was founded by a developer of Prado and it continues many of its conventions. Yii is very fast (leading in most benchmarks) and extensible, modular, and strictly object oriented. It has a rich set of features and decent documentation. It uses no special configuration or templating language, so you don't have to learn anything apart from object-oriented PHP to use it. Also, unlike many other frameworks, it follows pure MVC architecture with data being sent directly from Model to View.

Akelos

Started: 2006

License: LGPL

PHP versions: 4 or 5

Its logo is shown in Figure 1-12. Website: `http://www.akelos.org`, `http://github.com/bermi/akelos`

FIGURE 1-12: Akelos 2 logo

While all PHP frameworks are more or less inspired by Ruby on Rails, Akelos aims to be its direct port. It is focused on internationalization (provides multilingual *models* and *views* as well as Unicode support without extensions) and can run on low-cost shared hostings (that's why it has support for PHP4).

The author of Akelos announced the completely rewritten Akelos 2. It drops support for PHP4 and uses autoloading and lazier strategies for loading functionality. Its hallmarks will be advanced routing methods and strong REST orientation (REST is described in Chapter 12). It is to be released in late 2010 and it looks very promising.

Seagull

Started: 2001

License: BSD

PHP versions: 4.3.11+

FIGURE 1-13: Seagull logo

Its logo is shown in Figure 1-13. Website: `http://seagullproject.org`

Seagull is a true veteran among PHP frameworks — it was founded in 2001. Years of development made it solid, stable, and tested. It is no longer actively developed, so perhaps it is not the best choice when starting a new project, but there are still numerous successful applications that were built with it. It has contributed greatly to the development of all other PHP frameworks.

Qcodo

Started: 2005

License: MIT

PHP versions: 5.x

Its logo is shown in Figure 1-14. Website: `www.qcodo.com`

FIGURE 1-14: Qcodo logo

Qcodo is an MVC framework that excels in code generation from database design. It has a very powerful code generator that analyzes the structure of the data model, and creates PHP object code and also HTML pages for database manipulation. Perhaps this is not one of the more popular frameworks you are likely to hear about during a casual conversation, but several top institutions (including NASA) have applied it for their projects. Qcodo was created by Mike Ho of *QuasIdea Development* and is now developed by an active community. It also has a completely community-driven fork called Qcube.

Solar

Started: 2005

License: New BSD

PHP versions: 5.2+

Its logo is shown in Figure 1-15. Website: `http://solarphp.com`

FIGURE 1-15: Solar Framework logo

SOLAR stands for *Simple Object Library and Application Repository*. Its structure and naming conventions are similar to those of Zend Framework. One of the biggest differences is how you construct objects — all are created with a *unified constructor* and configured with an array in a config file. It has many helpful built-in example applications.

PHP On Trax

Started: 2007

License: GPL

PHP versions: 5.x

Its logo is shown in Figure 1-16. Website: `www.phpontrax.com`

As the name cleverly suggests, this framework was designed as an exact PHP copy of Ruby on Rails. At least it was meant to be because it still lacks many features and it is highly unlikely that it will finally realize this goal. It is just one of many good-looking frameworks that have eventually failed.

FIGURE 1-16: PHP on Trax logo

DESIGN PATTERNS IN WEB FRAMEWORKS

There are certain abstractions that can be transported between applications in order to make the development process faster. This section takes a closer look at these abstractions and the way they shape the web application frameworks.

It is not absolutely necessary to understand design patterns in order to start working with frameworks, so if you are bored, you can skip to the next chapter and come back here later. However, design patterns are fairly fundamental to these frameworks and application development as a whole, so we insist that you really come back here if you decide to skip this section now.

What Is a Design Pattern?

The definition of design pattern states that it is a *general solution to a commonly occurring problem in software design*. There is really not much more formal foundation because design patterns are a generally practical means that make up for a lack in formal mechanisms. Most often they are created when programming languages do not provide abstract mechanisms that become undeniably useful during the development of real-world applications.

A good analogy for design patterns is the game of chess. A novice player needs just to know the rules. It's like learning the basic syntax of a programming language. Still, knowing how a bishop moves doesn't make you a successful chess player, just like knowing how to open braces doesn't make you a PHP programmer. Skilled players are able to predict a few moves forward and respond with a winning scheme. That's like an experienced programmer who can, in fact, produce working software.

As you begin to master the game of chess, you begin to see patterns emerging. You can barely glance at the chessboard to classify the situation into one of these patterns and provide a proven response, both for present and future risks. You can perceive these patterns just intuitively, or you may try to name them. It's the same with software design patterns: when you are truly proficient, you use them all the time. There is a good chance that you have used some of them without even knowing it.

Naming design patterns is not necessary, but is indeed good for two things. First is an aid for thinking with patterns, because when you name something abstract, it is much easier to implement it in practice. Then you may further analyze this pattern, draw diagrams of it, and take full advantage of it. And the other thing is that you can share your experience. Chess players love to talk about various openings and gambits, and programmers can learn a lot by exchanging knowledge of design patterns as well.

And even more important, if you want another programmer to add some functionality to a fixed class and then tell him to use the Decorator pattern, you can expect that it will be done the way you want it rather than with a random makeshift solution. Thus design patterns have a great potential for preventing future problems.

Model-View-Controller as the Main Structural Design Pattern

Web frameworks take advantage of most, if not all, design patterns. However, MVC is the absolute structural backbone of all frameworks. The main idea of MVC is dividing the application into three layers:

- **Model** — Represents the business logic of the application. It is more than just the raw data; the Model has to represent the structure of data with all relationships and dependencies. It may comprise one or more classes that correspond to logic objects of the application and provide an interface for manipulating them. The Model is the only layer that uses persistent storage. It should completely encapsulate all database connections. The model should also notify the View when its internal state changes, so the View can be refreshed.
- **View** — The output displayed to the user. The most important thing is that the View never modifies the application data; it only presents it. There may be multiple Views for the same data, such as traditional HTML, PDF, Flash, or WML for mobile devices. They should be interchangeable without modifying the other layers.
- **Controller** — The part of an application responsible for handling user interaction and taking all other actions. The Controller should be created with simplicity in mind — it should be the controlling part that uses methods provided by the Model and the View; it shouldn't do everything by itself.

Figure 1-17 illustrates the relations between the three layers.

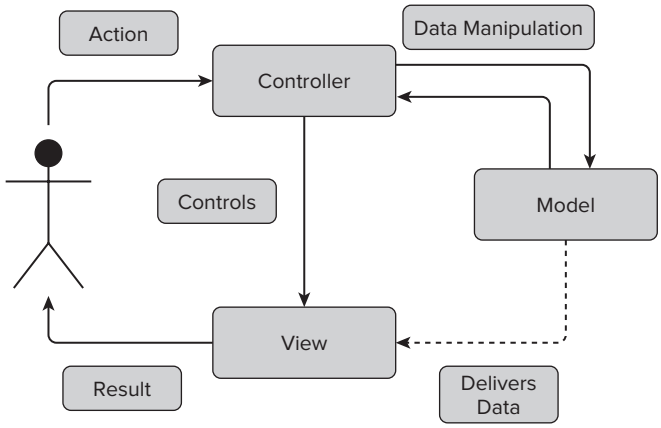

FIGURE 1-17: Model-View-Controller pattern

MVC versus MVP

MVC is an old design pattern, dating back to the 1979 work "Applications Programming in Smalltalk-80: How to use Model–View–Controller." by Trygve Reenskaug. Since that time, it was often used in non-web applications, mostly graphical user interfaces in compiled languages like C++ or Java. There it was easy and natural to implement an exact MVC pattern, but for web applications, it was somewhat modified.

Model-View-Presenter (MVP), shown in Figure 1-18, is a derivative of MVC. It is a three-tier application structure, where the Presenter acts as a middle layer between the View and the Model. The Presenter differs from the Controller in that it loads data from the Model and delivers it to the View.

Most so-called MVC frameworks follow the MVP pattern. While it is not bad itself because MVP seems even better suited to the task, this naming convention may be somewhat confusing. As long as MVP is derived directly from MVC, it is not a big problem, so in this book we will follow the names

conferred by the authors of the frameworks. So we will call all frameworks *Model-View-Controller*, even if the Controller does the majority of data-transferring work.

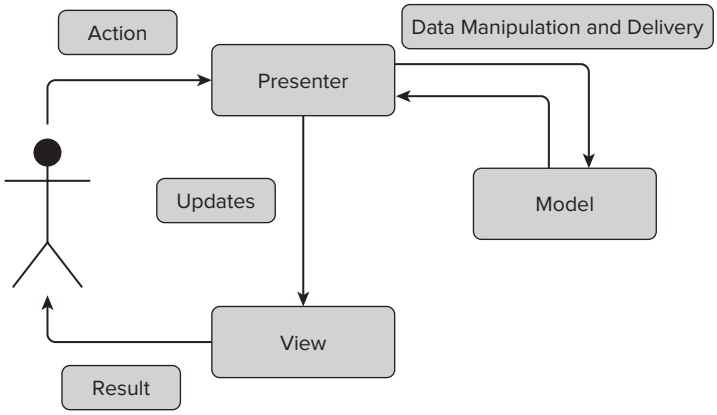

FIGURE 1-18: Model-View-Presenter pattern

Overview of Other Design Patterns

Design patterns can be divided into *creational*, *behavioral*, and *structural* patterns. Full description of all design patterns is well beyond the scope of this book, but you can find it in the most influential book on this subject: *Design Patterns*: *Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the *Gang Of Four*). However, we want to provide you with just a short overview of design patterns that are commonly used in web frameworks.

Singleton

This design pattern, which is so trivial it is often called an *antipattern*, is very useful. The purpose of the *Singleton* pattern is to ensure that a class has only one instance and to make this instance globally accessible. Whenever another object needs access to the Singleton, it calls a static, globally accessible function that returns reference to the single instance. You can see the structure of the Singleton in Figure 1-19.

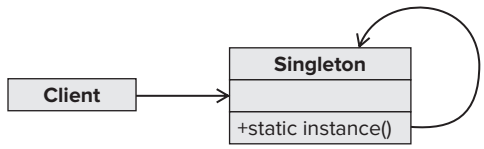

FIGURE 1-19: Singleton pattern structure

The trick behind *Singleton* is to make the instance and all its constructors *private*. So there is no way to demand creation of a `Singleton` class instance. How is the first and only instance created? The `instance()` method checks whether this object already exists; if not, it creates the single instance before returning it. Let's look at how this works with the PHP code.

```
<?php
class CarSingleton {
    private $make = 'Dodge';
    private $model = 'Magnum';
    private static $car = NULL;
```

```
    private static $isRented = FALSE;
    private function __construct() {
    }
    static function rentCar() {
        if (FALSE == self::$isRented ) {
            if (NULL == self::$car) {
                self::$car= new CarSingleton();
            }
            self::$isRented = TRUE;
            return self::$car;
        } else {
            return NULL;
        }
    }
    function returnCar(CarSingleton $carReturned) {
        self::$isRented = FALSE;
    }
    function getMake() {return $this->make;}
    function getModel() {return $this->model;}
    function getMakeAndModel() {return $this->getMake().' '.$this->getModel();}
}
?>
```

code snippet /singleton/CarSingleton.class.php

The class in the preceding code is a Singleton representing one concrete specimen of a Dodge Magnum car in a car rental business. The `__construct()` function is the constructor of this class. Note that it is set to `private` to prevent usage from outside of the class. The double underscore indicates that `__construct()` is one of the *magic functions* in PHP (special functions provided by the language), and declaring the constructor in a class will override the default one.

`CarSingleton` does provide an interface for renting and returning the car as well as pretty obvious getters. The `rentCar()` function checks first whether the car is already rented. This is not part of the Singleton pattern, but is important for the logic of our example. If the car wasn't rented, the function checks if the `$car` variable is `NULL` before returning it. If it equals `NULL`, it is constructed before the first use. Thus, `rentCar()` corresponds to the `instance()` method of the design pattern.

The `Customer` class in the following example represents a person who uses the services of the car rental business. He can rent the car (there is only one), return it, and tell the *make* and *model* of the car, provided that he drives it at the moment.

```
<?php
include_once('CarSingleton.class.php');
class Customer{
    private $rentedCar;
    private $drivesCar = FALSE;
    function __construct() {
    }
    function rentCar() {
        $this->rentedCar = CarSingleton::rentCar();
        if ($this->rentedCar == NULL) {
            $this->drivesCar = FALSE;
```

```
        } else {
            $this->drivesCar = TRUE;
        }
    }
    function returnCar() {
        $this->rentedCar->returnCar($this->rentedCar);
    }
    function getMakeAndModel() {
        if (TRUE == $this->drivesCar ) {
            return 'I drive '.$this->rentedCar->getMakeAndModel().' really fast!';
        } else {
            return "I can't rent this car.";
        }
    }
}
?>
```

code snippet /singleton/Customer.class.php

We can test these classes with the following code. It creates two customers, who both want to rent the car at the same time. But the second one will have to wait until the car is returned.

```
<?php
include_once('Customer.class.php');
$Customer_1 = new Customer();
$Customer_2 = new Customer();
echo 'Customer_1 wants to rent the car. <br />';
$Customer_1->rentCar();
echo 'Customer_1 says: ' . $Customer_1->getMakeAndModel() . '<br />';
echo '<br />';
echo 'Customer_2 wants to rent the car. <br />';
$Customer_2->rentCar();
echo 'Customer_2 says: ' . $Customer_2->getMakeAndModel() . '<br />';
echo '<br />';
$Customer_1->returnCar();
echo 'Customer_1 returned the car.<br />';
echo '<br />';
echo 'Customer_2 wants to rent the car. Again.' . '<br />';
$Customer_2->rentCar();
echo 'Customer_2 says: ' . $Customer_2->getMakeAndModel() . '<br />';
echo '<br />';
?>
```

code snippet /singleton/Test.php

The output of this code will look like this:

```
Customer_1 wants to rent the car.
Customer_1 says: I drive Dodge Magnum really fast!

Customer_2 wants to rent the car.
Customer_2 says: I can't rent this car.

Customer_1 returned the car.
```

```
Customer_2 wants to rent the car. Again.
Customer_2 says: I drive Dodge Magnum really fast!
```

The Singleton pattern is used often in other design patterns such as Prototype, State, Abstract Factory, or Facade. Apart from that, it can be used in all classes where you need a single instance with global access, but there is no way to assign it to another object, and perhaps you can also benefit from initialization on the first use. Be wary, though, because it is easy to overuse Singletons, and they may be dangerous, just like global variables. Another problem with Singletons is that they carry their state throughout the execution of the program, which seriously harms unit testing. Some experts even argue that Singleton is a bad idea and it generally should be avoided.

Frameworks use Singletons for various reasons. One of them is storing user data for security purposes. You want to have a single instance of a user that holds authentication data and make sure that no second instance can be created. This approach is represented, for example, by the `sfGuard` class of Symfony.

Prototype

The *Prototype* pattern is useful when you need the flexibility of parameterized object creation and when you want to avoid using the `new` operator. Object creation is done here by creating a parent class with an abstract `clone()` method and a few subclasses implementing `clone()`. Each of these subclasses comes with one instantiated Prototype object, which clones itself when you call for a new instance. This results in easiness and flexibility of object creation — you don't have to hard-wire the concrete subclass name in your code. Instead you can pass the name of the class as a string or reference to the appropriate Prototype.

This pattern also greatly supports deep copying of objects. Instead of cloning the Prototype, you can clone an existing object, receiving an exact copy as the result. You can even copy objects from a container with mixed objects of various subclasses. The only requirement is that they implement the `clone()` interface. Copying objects this way is much faster than creating objects with `new` and assigning values. A general diagram of this pattern is shown in Figure 1-20.

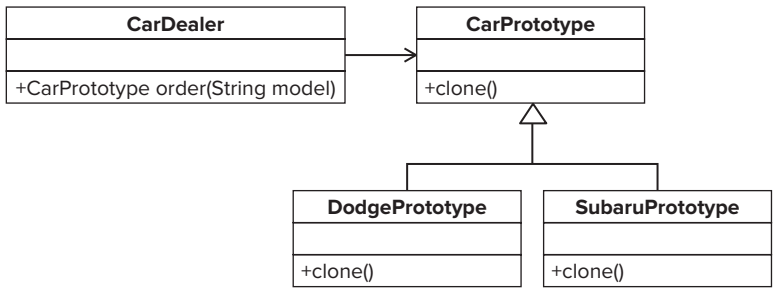

FIGURE 1-20: Prototype pattern structure

PHP has another *magic function*: `__clone()` does most of the work for you. All you have to do in the following example is to create an abstract `CarPrototype` class and subclasses for different producers. The `__clone()` function is declared `abstract`, so subclass methods are used by default when this method is called.

```
<?php
abstract class CarPrototype {
    protected $model;
    protected $color;
    abstract function __clone();
    function getModel() {
        return $this->model;
    }
    function getColor() {
        return $this->color;
    }
    function setColor($colorIn) {
        $this->color= $colorIn;
    }
}
class DodgeCarPrototype extends CarPrototype {
    function __construct() {
        $this->model = 'Dodge Magnum';
    }
    function __clone() {
    }
}
class SubaruCarPrototype extends CarPrototype {
    function __construct() {
        $this->model = 'Subaru Outback';
    }
    function __clone() {
    }
}
?>
```

code snippet /prototype/CarPrototype.class.php

Cars are quite an accurate example here, because in real life a prototype is created by a manufacturer and then different models are based on this prototype and filled with unique features. The following code tests the preceding classes. First, it creates two Prototype objects as showcase cars and then clones one of them to serve the customer. Then the color can be picked by the uniform interface.

```
<?php
include_once('CarPrototype.class.php');
$dodgeProto= new DodgeCarPrototype();
$subaruProto = new SubaruCarPrototype();
echo 'Which car do you want? <br />';
$customerDecision = 'Subaru';
if( $customerDecision == 'Subaru' ){
   $customersCar = clone $subaruProto;
} else {
   $customersCar = clone $dodgeProto;
}
echo $customersCar->getModel().'<br />';
echo 'What color do you want?<br />';
$customersCar->setColor('red');
```

```
echo 'Fine, we will paint your '.$customersCar->getModel().
     '  '.$customersCar->getColor().'.<br />';
?>
```

code snippet /prototype/Test.php

The previous code will result in the following messages:

```
Which car do you want?
Subaru Outback.
What color do you want?
Fine, we will paint your Subaru Outback red.
```

The Prototype pattern is used commonly in different modules of frameworks. An example can be nesting of forms within forms in Symfony or the `AppController` class of CakePHP.

Decorator

Subclassing is a great mechanism, but it has some serious limitations. Suppose that you want to produce a car. You put all your effort into designing a good yet affordable standard model of the car. It is a complete design that defines the *look and feel* of this model and it is a reference for any possible modifications. Then you seek to provide some optional equipment that improves the quality of the car and adds some new functionalities. For example, it may be all-wheel drive instead of front-wheel drive. It may be automatic transmission instead of manual. The car may also come in different trim levels with electric leather seats, sunroof, better audio, or GPS satellite navigation. However the basic interface remains the same — you can drive this car and feel good doing it.

When you face such alternatives, the number of possible combinations rises really fast. Figure 1-21 shows some combinations for just three improvements, described as inheritance hierarchy.

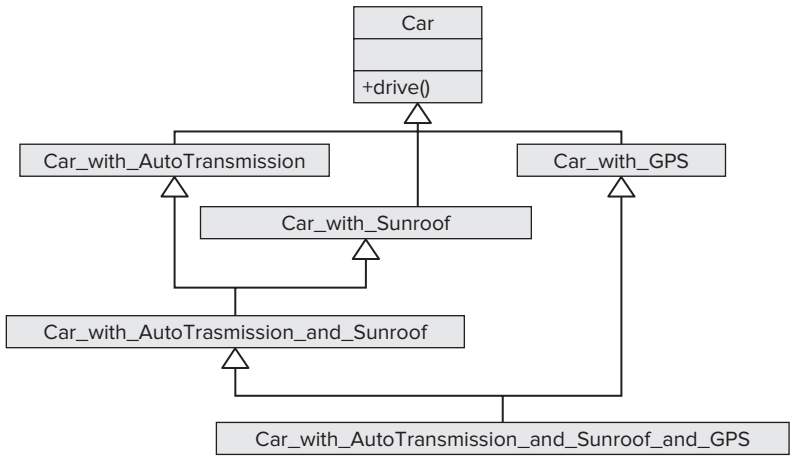

FIGURE 1-21: Nasty inheritance hierarchy

The answer to this problem is the *Decorator* pattern. The Decorator is a class that shares the interface with the decorated class (in our example, it is the basic car). It encapsulates an instance of the

decorated object and extends its responsibilities dynamically. It is like putting a gift into a solid box and then wrapping it with colorful paper — it is still a gift, but durable and decorated. The inheritance structure of the Decorator pattern is presented in Figure 1-22.

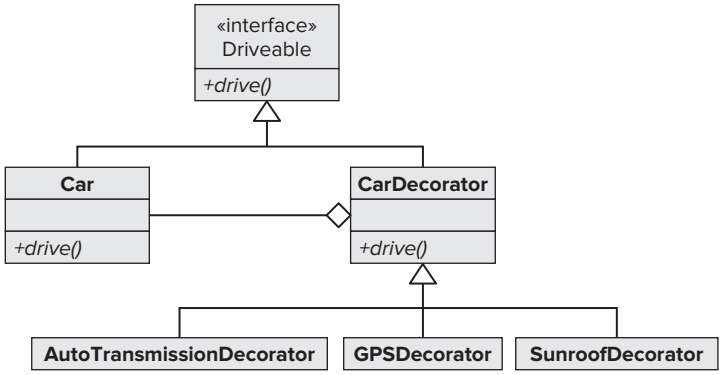

FIGURE 1-22: More reasonable inheritance hierarchy with Decorator

You can put the decorated object into other Decorators without limitations. This way you can add as many optional modules as you wish. Decorators can have their own inheritance hierarchy, and within this hierarchy they encapsulate the core object recursively.

The code below creates a standard `Car` class without optional equipment.

```
<?php
class Car{
  public $gearMessage = 'Remember to shift up.';
  public $comfortMessage = 'standard.';
  function drive() {
    return 'Accelerating ' . $this->gearMessage .
           ' Driving comfort is ' . $this->comfortMessage;
  }
}
?>
```

code snippet /decorator/Car.class.php

The following classes are responsible for extending the functionality of the car. The first one, `CarDecorator`, is the first level of wrapping. It stores the `$car` variable and a copy of `$comfortMessage`. This variable will be changed by a Decorator, so we create a copy to avoid changing the original `$car` object. On the other hand, `$gearMessage` is changed internally. The `drive()` function is also subclassed to use the proper variables `$car->model` and `$this->gearMessage` because we want to access the core object here, but `$this->comfortMessage` because we want to use the amended value.

Second-level Decorators wrapping the `CarDecorator` are used to install optional components, as shown below. `AutomaticTransmissionDecorator` installs the `$gearMessage` directly into the core `$car`, but `GPSDecorator` is installed into the `CarDecorator` instead. Note that all decorators share the common interface and additionally provide specific installers.

```
<?php
    class CarDecorator {
      protected $car;
      protected $gearMessage;
      protected $comfortMessage ;
      public function __construct(Car $car_in) {
        $this->car = $car_in;
        $this->comfortMessage = $car_in->comfortMessage;
      }
      function drive() {
        return 'Accelerating. ' . $this->car->gearMessage .
               ' Driving comfort is ' . $this->comfortMessage;
      }
    }

    class AutomaticTransmissionDecorator extends CarDecorator {
      protected $decorator;
      public function __construct(CarDecorator $decorator_in) {
        $this->decorator= $decorator_in;
      }
      public function installAutomaticTransmission(){
        $this->decorator->car->gearMessage = 'Auto transmission shifts up.';
      }
    }
    class GPSDecorator extends CarDecorator {
      protected $decorator;
      public function __construct(CarDecorator $decorator_in) {
        $this->decorator= $decorator_in;
      }
      public function installGPS(){
        $this->decorator->comfortMessage= 'very high.';
      }
    }
?>
```

code snippet /decorator/CarDecorator.class.php

We can test these classes with the following code.

```
<?php
  include_once('Car.class.php');
  include_once('CarDecorator.class.php');
  $car = new Car();
  $decorator = new CarDecorator($car);
  $transmission = new AutomaticTransmissionDecorator($decorator);
  $gps = new GPSDecorator($decorator);
  echo 'Driving standard car: <br />';
  echo $car->drive().'<br />';
  $transmission->installAutomaticTransmission();
  $gps->installGPS();
  echo 'Driving fully decorated car: <br />';
  echo $decorator->drive() . '<br />';
```

```
  echo 'Driving the car without decoration: <br />';
  echo $car->drive() . '<br />';
?>
```

code snippet /decorator/Test.php

And the result will be the following:

```
Driving standard car:
Accelerating. Remember to shift up. Driving comfort is standard.
Driving fully decorated car:
Accelerating. Auto transmission shifts up. Driving comfort is very high.
Driving the car without decoration:
Accelerating. Auto transmission shifts up. Driving comfort is standard.
```

First we call the basic `Car` model. Next we install the optional equipment and call the `drive()` function of the `CarDecorator`. Finally we choose to drive the car not using the Decorator wrapping. Note that after calling the `$car` then, its transmission is still automatic. That's because the Decorator changed it permanently.

Going back to frameworks, the Decorator pattern is used among others for layouts and templates. It is very useful for adding optional visual components or extending the user interface when new widgets are needed. An example may be adding scrollbars when user input exceeds the field area.

Chain of Responsibility

The three preceding design patterns concerned object creation and inheritance structure. *Chain of Responsibility* is a pattern of another kind, because it applies to the behavior of objects. Its main intent is to decouple the *sender* of a request from its *receiver*. Let's see how it works with an automotive example.

Imagine that there is an emergency on the road and you need to quickly stop the car. In other words, `stop` is the emitted request. In most cases, hitting the brake pedal is a sufficient solution, but there are rare cases when you find the brakes broken; that's when *Chain of Responsibility* comes in handy. If brakes cannot handle the request, they pass it to the handbrake. If for any reason the handbrake is broken, too, and you are going to hit the obstacle, at least airbags should open potentially saving your life. Airbags are the most *generic solution* to most road emergencies. They are less preferred than more specialized solutions (braking, evading), but still better than nothing if those maneuvers fail. It's the same with your applications — it is better to give the request a chain of potential handlers, as shown in Figure 1-23, instead of letting it fail without even an error message.

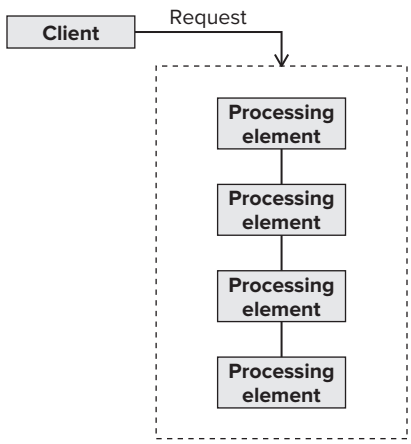

FIGURE 1-23: Chain of Responsibility as a response to a request

So, how do you create such Chain of Responsibility? The main idea of this pattern is to process a request by a list of consecutive handlers to avoid any hard-wired mappings. The initial client holds a reference only to the first element in the chain of handlers. Then each handler holds a reference to the handler afterward. The last handler must always accept the request to avoid passing it to a `NULL` value.

A good class structure supporting this behavioral pattern is shown in Figure 1-24. It consists of a parent `Handler` class that calls the `handle()` method to delegate the request to the next concrete handler `nextHandler`. This `Handler` class is subclassed by concrete handlers that try to do something with the request; if they fail, they call the `handle()` method of their superclass.

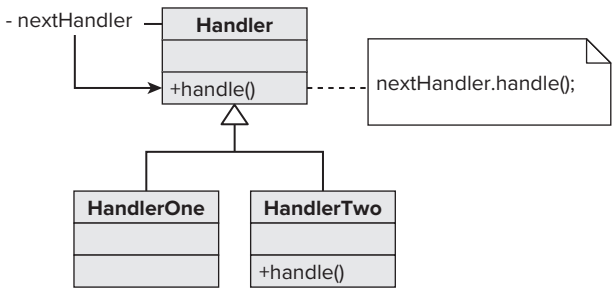

FIGURE 1-24: Chain of Responsibility pattern structure

Chain of Responsibility is commonly used for filters. One example of filtering is when a user request is being processed. First it checks whether the given controller exists or not. If it doesn't exist, a *404 error* is displayed. If it does exist, the request is passed to the controller, which handles it further. It checks whether a user tries to access an unsecured page; if it's true, it redirects the request to an SSL-secured page. Then it is checked for authentication, and so forth.

State

Sometimes you want a component to behave differently for various possible states of the application. First, define an abstract `State` class, which is a common interface for various `ConcreteStates`. All states provide a `handle()` method that provides various behaviors of your component. The `Context` class is the core class that wraps a `ConcreteState` *`state`* object. This design pattern makes sense when `Context` is a complete class that also provides state-independent functionalities. Otherwise, simple subclassing of `Context` would be more efficient.

`Context` calls the `state->handle()` method when processing its own requests. `Context` also has methods for switching between States. Depending on which `ConcreteState` the *`state`* variable holds, the `state->handle()` method provides different behaviors. This can be regarded as emulation of a partial type change at runtime. You can see a diagram of this pattern in Figure 1-25.

The State pattern, although rather simple, is very useful for application development. One example is database connection — the database abstraction layer may change its behavior depending on the current connection state. Another example can be the state of a transaction in an online store — the

application may display different pages depending on which steps are needed to complete the transaction.

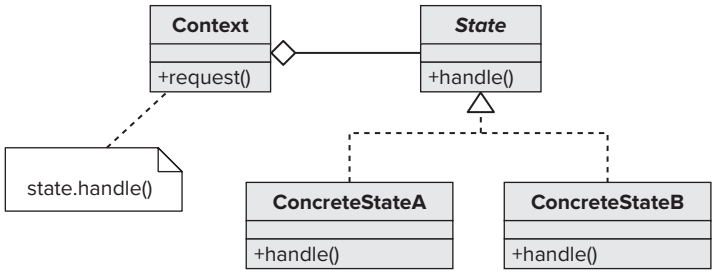

FIGURE 1-25: State pattern structure

Iterator

There are many kinds of aggregate objects and many ways to traverse them. One simple example is an array traversed by consecutive integers supplied to the *array operator*. To print out a five-element `myArray`, you could use the following:

```
for ($i=0;$i<=4;$i++) {
echo $myArray[$i];
}
```

However, this solution is not at all elegant. First of all, you have to take care of `i` variable values. PHP is not C/C++, so it is not catastrophic to call for `myArray[100]` here — it will not return random trash from memory. However, it is still easy to skip some values with hard-wired ranges. Another problem is that such an approach exposes the underlying representation of this aggregation. It makes the traversal procedure dependent on this specific representation and thus is not reusable. Object-oriented programming aims to encapsulate the internal structure of aggregate objects and provide a uniform, safe, and useful interface like this one provided by PHP:

```
interface Iterator{
function current(); // Returns the value of element under current key
function key(); // Returns the current key
function next(); // Moves the internal pointer to the next element
function rewind(); // Moves the internal pointer to the first element
function valid(); // Returns true if the element under current key is valid
}
```

Now every class implementing this interface can use the `foreach` structure. The following snippet of code produces the same output as the previous `for` loop:

```
foreach ($myArray as $value) {
echo $value;
}
```

The abstraction behind this mechanism is the *Iterator* design pattern, pictured in Figure 1-26. The `Client` application has access to two abstract classes: `Collection`, which is the aggregate object interface, and `TraversalAbstraction`, which is created by a corresponding `Collection`. The underlying concrete collections can be as different as a *List* and a *Map*, yet corresponding methods

of traversal can be produced for both of them. When `Client` calls the `next()` method, different ordering algorithms are executed for *List* and for *Map*, but in both cases a subsequent element is found.

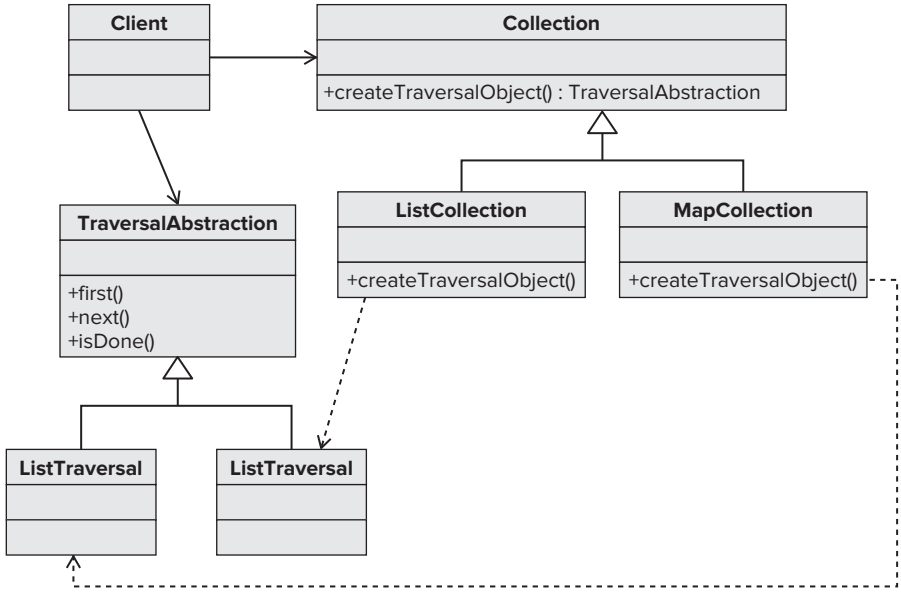

FIGURE 1-26: Iterator pattern structure

In web frameworks the Iterator pattern is used mainly for pagination. You need a uniform interface to divide web content into adequate pieces, turn them into separate web pages, and then traverse through them.